AD-Suite - A Test Suite for Algorithmic Differentiation

M. Narayanamurthi*, T. Bosse†, S. Narayanan† and P. Hovland†

March 29, 2016

1 Introduction

Algorithmic differentiation (AD) is a widely accepted methodology to obtain derivatives of scientific code for use in optimization, integration methods, and sensitivity analysis. Currently about 60 AD tools are listed on `autodiff.org` to compute sensitivity information. Several years have been invested in the research and development of popular implementations such as ADOL-C, CppAD, OpenAD, and Tapenade. In most cases, however, the test code that comes packaged along with the tools comprises either toy or academic examples that are far removed from real-life applications. In general testing scientific software is difficult. A common reason is the lack of separation between theory and code in a scientist's mind that limits testing to a mere verification of theory [7].

The lack of good test codes for AD in particular can be attributed to a couple of factors. One is that tools are written in and for different programming languages — C/C++, Python, Java and Fortran. The second is that AD is usually implemented by using operator overloading or source transformation and requires special formatting of the input code. These two factors contribute to the need for changes in the original simulation code or even recoding of the whole program in order to apply the developed AD software and get appropriate results. Drawing inspiration from projects that attempt to solve the problem of testing scientific software in other fields such as [1], [3], [4], [5], and [6], we propose to create AD-Suite: a test suite for AD and a classification that describes the applications included in that suite. To the best of our knowledge, there exists neither a classification of applications nor a test suite for AD. Although, projects such as `SifDEC` provide the ability to generate input for AD tools, they deal mostly with nonlinear optimization and hence narrow the scope for an AD classification suite; see, for example, [2], where most or all of the problems within the CUTEr test set have sparse derivatives. An additional barrier is that one needs to create a `Sif`-to-any-programming-language source transformation tool in order to generate test cases in different languages and the required special format (and this of course has to be tested independently). Also, most users are understandably unwilling to learn a new format and recode their application in order to apply AD.

Instead, we aim to crowd-source codes in multiple languages, spanning several application areas and having code structure that is typical in practice. In particular, we hope that AD users will actively contribute to the test suite with minimal additional effort, because further advancement of open-source AD tools is highly dependent on the variety and richness of the provided applications and examples. The examples also yield instructive templates for other interested users who want to apply AD. For AD developers, AD-Suite will be useful in a wide range of scenarios. For example, available applications will enable AD developers to better understand the needs of the users, anticipate future research areas; and design appropriate drivers. Moreover, it can be used to validate the correctness of a new implementation and compare the results. It also allows for performance comparison with other tools and different approaches on a "suitable" set of problems.

We expect AD-Suite to be publicly available on `autodiff.org` and contain for each problem all necessary information. As part of this effort, we envision a webform where we will collect the information along with each submission in a standard format. Through this endeavor, we hope to strengthen the AD community by bringing together AD developers, industrial partners, and users of AD.

2 AD-Suite

From a mathematical point of view, most scientific codes can be thought of as a sufficiently smooth function $F : \mathbb{R}^n \times \mathbb{R}^q \to \mathbb{R}^m$ with output $y = F(x, p) \in \mathbb{R}^m$ that depends on some parameter $p \in \mathbb{R}^q$ at a given base-point $x \in \mathbb{R}^n$. Depending on the application, these functions usually have special characteristics and/or a certain structure, which is reflected in their implementation. For example, in regression analysis the function F represents a code evaluating the objective function of the optimization problem

$$\min_{x \in \mathbb{R}^n} F(x, p) = \sum_{i=1}^{r} \|f(x, d_i) - m_i\|. \tag{1}$$

*Corresponding Author, Virginia Tech, US, `maheshnm@vt.edu`
†Argonne National Laboratory, IL, US

The objective function minimizes the sum of $r \in \mathbb{N}$ residuals, in a given norm, between the observed measurements $m_i \in \mathbb{R}^k$, and a model function $f : \mathbb{R}^n \times \mathbb{R}^l \to \mathbb{R}^k$ for some corresponding input $d_i \in \mathbb{R}^l$. In other words, $x \in \mathbb{R}^n$ has to be chosen such that f fits the data $p = (d_i, m_i)_{i=1}^r$ in an optimal way. Obviously, parts of this function/code can be evaluated in parallel, and the same holds true for its derivatives. A different class of examples, which typically arise in time integration methods, comprises functions that represent a repeated composition

$$F(x, p) = F_r(F_{r-1}(\dots F_2(F_1(x, p), p), \dots, p), p) \tag{2}$$

of functions $F_i : \mathbb{R}^{n_i} \times \mathbb{R}^q \to \mathbb{R}^{m_i}$ with $m_{i-1} = n_i$, $i = 2, \dots, r$, such that each successive stage of the computation is dependent on the previous stage.

These structures need to be taken into account for an efficient evaluation of the sensitivity information for F by some AD tool. Other features include the smoothness of F, the sparsity of the derivatives, and the number of (in-)dependent variables (see Table 1). In fact, most AD tools have specialized drivers that allow an efficient treatment for the different classes of problems. For example, ADOL-C provides a checkpointing algorithm to efficiently compute adjoints of functions F that are given by a repeated composition [8]. Naturally, a successful implementation of such drivers requires suitable test examples in order to validate the computational results and get a measure of the progress from the latest developments.

2.1 AD Classification

In addition to collecting code samples from the AD community, we would like to define an extensible standard that can be used to categorize each application. This standard will help the developers find appropriate test examples and allow AD users to choose the correct AD software or driver. Therefore, we follow the CUTEr example and assign each problem an intuitive label that consists of $5 + 3 + (1)$ alpha-numeric entries that are separated by hyphens using the ordered categories and code features given in Table 1. In detail, we propose to use the following (ordered) syntax.

$$\underbrace{}_{\text{Smoothness}} - \underbrace{}_{\text{Code Feature}} - \underbrace{}_{\text{Non-linearity}} - \underbrace{}_{\text{Sparsity}} - \underbrace{}_{\text{Parallel}} - \underbrace{\_\_-\_\_-\_\_}_{\text{Dimensions n,m,p}} - \underbrace{}_{\text{Special}}$$

For example, the label `C2-L-N-S-S-100-10-4` can be used to denote a twice differentiable problem with a loop structure that is nonlinear and has sparse derivatives similar to that given in (2). Its evaluation is serial, with 100 independent and 10 dependent variables and 4 parameters. The (optional) special feature can be used to provide user-specified information about a problem in more detail depending on its underlying structure or implementation using the given classification codes. For example, if the problem has a loop consisting of 99 iterations, one could label the previous example by `C2-L-N-S-S-100-10-4-L99`. In particular, every code feature with the exception of the dimensions, which is not yet listed in Table 1, should be a capital letter to allow for multiple features. It can be followed by one or more lower-case letters and numerical values, as was done in the given example to indicate its smoothness (`C2`), or the special feature to describe the number of iterations for the loop (`L99`); that is, each entry can consist of one or more entries of the form [A-Z][a-z0-9] using Sed syntax. This allows the convention to be more extensible and stay consistent. For example, the first example (1) gives rise to Lipschitz continuous functions (Lc0) with a nested structure (N) that require the computation of derivatives and the solution of linear problems (S) within the code itself. Hence, an appropriate label would be `Lc0-NS-N-D-P-5-1-99999` if the nonlinear function with dense derivatives has 5 independents, 1 dependent variable, and a large number of parameters.

Separately, AD developers may also use the label to provide information about the tool capability. For example, `C2-L-N-*-S-500-500-*` may represent an AD tool with checkpointing capability (L) that can efficiently handle at least twice differentiable (C2) nonlinear functions (N) with a serial loop structure (S), which could have sparse or dense derivatives (*), up to 500 (in)dependent variables, and an arbitrary amount of parameters (*).

Category	**Code features**
Smoothness of F	Lipschitz continuous (C0), C^1 (C1), C^2 (C2),...,C^∞ (CI)
Source Code Features	Loop (L), (non-)linear Solver (S), Cross-derivative (C), Nested Derivative (N),...
Nonlinearity of F	Linear (L), Quadratic (Q), Rational (R), Nonlinear (N)
Sparsity	Sparse (S), Dense (D), Block-structure (B)
Parallel	Serial (S) , Parallel (P),
Dimensions	m-n-p: Independents (n), Dependents (m), Parameters (p)
(Special Feature)	(user-specified)

Table 1: Classification codes for different AD applications

2.2 Standard Test Problems for AD

We expect the AD-Suite to contain applications that have been successfully differentiated, along with supporting files (described below) and the correct classification code. Submissions to the suite will use the directory structure illustrated in Figure 1 and will contain the following information.

```
example...      The root directory for a problem contains the README, Makefile, eval_F.*,....
 ├─data...      This directory contains subdirectories for different test scenarios.
 │  ├─data1...  This directory contains the information for one specific test case.
 │  ├─data2...  This directory contains the information for another specific test case.
 │  └─...
 ├─doc...       This directory contains the documentation for the example code.
 ├─src_code...  This directory holds all the source code files for the example code.
 └─src_deriv... This directory holds all the files for the differentiated code or used drivers.
```

Figure 1: Template of the directory structure for an example from the AD-Suite

Basic instructions and information The root folder must contain a `README` file with the classification code, the name, and a short description of the problem. Additionally, the file must include information about the authors, the required third-party packages, the language it is coded in, and the tool it has been differentiated with.

Data for testing purpose At least one set of numerical reference values for the independent variables x, the parameters p, and the result $y = \mathcal{F}(x) = F(x, p)$ must be provided, which can then be used to validate the correctness of the code in case of modifications, without running the original code. Each test case will be stored in a separate subdirectory of `data`. Within this folder, the reference values for x, p, and y must be saved in the three `csv`-files `x.csv`, `param.csv`, and `y.csv`, respectively. Furthermore, the directory must contain computed results of the differentiated code, namely, the derivatives of F with respect to x:

$$\mathcal{F}'(x), \quad \mathcal{F}''(x), \quad \mathcal{F}'(x)\,\bar{x}_1, \quad \bar{y}_1^\top\,\mathcal{F}'(x), \quad \mathcal{F}''(x)\,\bar{x}_1\,\bar{x}_2, \quad \ldots.$$

The files for the Jacobian/first derivative, Hessian/second derivative, and so on are simply called `deriv_1.csv`, `deriv_2.csv` ,.... The values for directions $\bar{x}_i$ and adjoint $\bar{y}_j$ are stored in the files `dir_i.csv` and `adj_j.csv`, respectively. The resulting numerical values for derivatives such as the Jacobian-vector product $\mathcal{F}'(x)\,\bar{x}_1$ are given in correspondingly named files (e.g., `deriv_1_dir1.csv`). Also, this folder must contain additional files for applications with a special structure; for example, one must store the intermediate values of each iteration i in the files `iter_i.csv` if the example involves a loop and allows for checkpointing.

Documentation and literature for the test Often, the documentation of scientific codes is limited because they are not meant to be used outside the research group that develops them. Usually, the documentation consists of only minor comments within the code of the form "this method implements formula XX in YY". Naturally, we do not expect that the developers will provide detailed documentation of the code solely for the purpose of inclusion in the test suite. We do expect, however, that cited literature in some standard format (typically `pdf`) will be provided in the folder `doc` to help other researchers and AD developers grasp the most important steps of the code.

Source code of the example The code must be completely contained within the directory `src_code`. Driver files, the data for the variable x, the parameter p, the directions $\bar{x}_i$, and the adjoints $\bar{y}_j$ with the corresponding output exist elsewhere. The driver files are in the root directory with a corresponding `Makefile` that includes for all third-party packages a path variable, which needs to be adapted such that the code (compiles and) executes. The necessary changes should be described in the `README` file. Each of the driver files must contain only one method that either reads one of the dimensions for the scenario t, the initial values for x or p, or evaluates $y = \mathcal{F}(x)$ with corresponding filenames, for example, `get_dim_n.X`, `get_x.X`, and `eval_F.X` for the language-specific extension `X`. The standard signature of the methods follows the input/output convention

$$\texttt{get\_dim\_n(}\ \underset{\downarrow}{n},\ \overset{\downarrow}{t}\ \texttt{)},\quad \texttt{get\_x(}\ \overset{\downarrow}{n},\ \underset{\downarrow}{x},\ \overset{\downarrow}{t}\ \texttt{)},\quad \texttt{save\_x(}\ \overset{\downarrow}{n},\ \overset{\downarrow}{x},\ \overset{\downarrow}{t}\ \texttt{)},\quad \texttt{eval\_F(}\ \overset{\downarrow}{n},\ \overset{\downarrow}{m},\ \overset{\downarrow}{q},\ \overset{\downarrow}{x},\ \underset{\downarrow}{y},\ \overset{\downarrow}{p},\ \underset{\downarrow}{flag}\ \texttt{)}.$$

In particular, the code must run the tth test scenario without errors or abortion after modifying the `Makefile` and calling `make && make test` t. Furthermore, the call `make test` t should also create the output file `y.csv` in the corresponding `data` subdirectory.

Differentiated code or corresponding driver files Similar to the original source, we expect that the differentiated code must be stored in the separate folder `src_deriv`. The AD tools used to generate the derivative code(s), along with the version (public release) or revision number (code repository), have to be specified in the `README` file. Again, the driver files for the differentiated code are in the root directory and contain only one method, for example, `deriv_1.X`, `deriv_1_dir.X`, and `adj_deriv_1.X` for the computation of the Jacobian, Jacobian-vector, and vector-Jacobian products, respectively. The standard signature of the methods follows the input/output convention

$$\texttt{deriv\_1(}\ \overset{\downarrow}{n},\ \overset{\downarrow}{m},\ \overset{\downarrow}{q},\ \overset{\downarrow}{x},\ \underset{\downarrow}{y},\ \underset{\downarrow}{\mathcal{F}'(x)},\ \overset{\downarrow}{p},\ \underset{\downarrow}{flag}\ \texttt{)},\quad \texttt{deriv\_1\_dir(}\ \overset{\downarrow}{n},\ \overset{\downarrow}{m},\ \overset{\downarrow}{q},\ \overset{\downarrow}{x},\ \overset{\downarrow}{\bar{x}_1},\ \underset{\downarrow}{y},\ \underset{\downarrow}{\mathcal{F}'(x)\bar{x}_1},\ \overset{\downarrow}{p},\ \underset{\downarrow}{flag}\ \texttt{)},\quad \ldots.$$

The differentiated code must compile and run without errors or abortion, for example, after calling `make && make deriv_1` t to evaluate the Jacobian of test scenario t. Additionally, the output file `deriv_1.csv` must be generated in the `data` subdirectory.

Testing information Besides the data for the validation of the code and its derivatives, other information may be of interest to a user, such as the sparsity ratio for each computed derivative. Of special importance are performance metrics for comparing different AD tools applied to a considered test example. Of course, these performance metrics should be measured by normalized values because simple run-time measurements are overly influenced by other factors such as the programming language and the system architecture. More independent estimates for the performance of an AD tool seem to be the averaged run-time ratios for each task w.r.t. the time required for evaluating the original function itself:

$$\frac{\text{Time trace}\,[\mathcal{F}(x)]}{\text{Time eval}\,[\mathcal{F}(x)]}, \quad \frac{\text{Time eval}\,[\mathcal{F}'(x)\,\bar{x}_1]}{\text{Time eval}\,[\mathcal{F}(x)]}, \quad \frac{\text{Time eval}\,[\bar{y}_1^\top\,\mathcal{F}'(x)]}{\text{Time eval}\,[\mathcal{F}(x)]}, \quad \ldots$$

Similarly, one can define a normalized measure for the ratio of peak/average memory consumption with respect to the peak/average memory consumption for only the function evaluation. For completeness, the software and hardware platform used for obtaining the metrics should be reported. All this information will be collected in an editable simple `csv`-file on `autodiff.org` with additional information that could, for example, include the number of used parallel processes or the checkpointing pattern, density, or ratio if applicable. The file can be used for state-of-the-art performance plots without the necessity of redoing all the work and testing.

3 Conclusions and further work

We have started collecting a small number of "realistic" problems, which we will provide to the AD community for testing. The examples are given in the proposed format and are used to describe the "standard" format in more detail. Moreover, we will try to formulate a suitable extension of the standard to provide additional information such as the underlying discrete graph structure for the test-set problems that could be used by other AD developers to improve their algorithms. We envision having AD-Suite become an interactive part of the AD website `autodiff.org` along with a small set of minitools, which could be used, for example, to create performance plots.

Acknowledgments

This material was based upon work supported the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. CUTE: constrained and unconstrained testing environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, Mar. 1995.

[2] Torsten Bosse and Andreas Griewank. *Recent Advances in Algorithmic Differentiation*, chapter The relative cost of function and derivative evaluations in the CUTEr test set, pages 233–240. Springer, Berlin, Heidelberg, 2012.

[3] Nicholas I. M. Gould, Dominique Orban, and Philippe L. Toint. CUTEr and SifDec. *ACM Transactions on Mathematical Software*, 29(4):373–394, Dec. 2003.

[4] Nicholas I. M. Gould, Dominique Orban, and Philippe L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, 60(3):545–557, Aug. 2014.

[5] Raghu Pasupathy and Shane Henderson. A testbed of simulation-optimization problems. In *Proceedings of the 2006 Winter Simulation Conference*. Institute of Electrical & Electronics Engineers (IEEE), Dec. 2006.

[6] Raghu Pasupathy and Shane G. Henderson. SimOpt: A library of simulation optimization problems. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*. Institute of Electrical & Electronics Engineers (IEEE), Dec. 2011.

[7] Rebecca Sanders and Diane Kelly. Dealing with risk in scientific software development. *IEEE Softw.*, 25(4):21–28, Jul. 2008.

[8] Andrea Walther and Andreas Griewank. Getting started with ADOL-C. In Uwe Naumann and Olaf Schenk, editors, *Combinatorial Scientific Computing*. Chapman-Hall, 2012.